

High-Performance GPU-to-CPU Transpilation and Optimization via High-Level Parallel Constructs

Anonymous Author(s)

Abstract

While parallelism remains the main source of performance, architectural implementations and programming models change with each new hardware generation, often leading to costly application re-engineering. Most tools for performance portability require manual and costly application porting to yet another programming model.

We propose an alternative approach that automatically translates programs written in one programming model (CUDA), into another (CPU threads) based on Polygeist/MLIR. Our approach includes a representation of parallel constructs that allows conventional compiler transformations to apply transparently and without modification and enables parallelism-specific optimizations. We evaluate our framework by transpiling and optimizing the CUDA Rodinia benchmark suite for a multi-core CPU and achieve a 58% geomean speedup over handwritten OpenMP code. Further, we show how CUDA kernels from PyTorch can efficiently run and scale on the CPU-only Supercomputer Fugaku without user intervention. Our PyTorch compatibility layer making use of transpiled CUDA PyTorch kernels outperforms the PyTorch CPU native backend by 2.7x.

1 Introduction

The end of single-core performance scaling means that parallelism and domain-specificity are now the main sources of efficiency increases. Supercomputer architects compete in ingenuity to support compute- and memory-intensive applications from physics simulations to machine learning. The latest and fastest supercomputer, Fugaku, is based exclusively on A64FX CPUs that, unlike commodity CPUs, provide support for high-bandwidth memory access and energy efficiency comparable to that of recent GPUs [42].

However, efficient and productive use of such computers for practical applications is challenging as recent frameworks and high-performance libraries have been developed with Nvidia GPUs in mind. For example, attempts to port PyTorch [38] to A64FX have met multiple challenges. The “native” default CPU PyTorch backend provides only naïve versions for critical kernels, such as 2D convolution implemented as six nested loops. Intel’s oneDNN [24] unsurprisingly performs poorly for Arm CPUs since it is tailored

for commodity CPUs without high-bandwidth memory. Fujitsu’s customized oneDNN [17] is better tuned, but not universally competitive with GPUs.

Many non-library approaches for performance portability have been proposed and include language extensions (e.g., OpenCL [12], OpenACC [22]), parallel programming frameworks (e.g., Kokkos [3]), domain-specific languages (e.g., SPIRAL [15], Halide [41] or Tensor Comprehensions [56]). All of these approaches still require legacy applications to be ported, and sometimes entirely rewritten, due to differences in the language, or the underlying programming model.

We explore an alternative approach based on a fully automated compiler that takes code in one programming model (CUDA) and produces a binary targeting another one (CPU threads). While GPU-to-CPU translation has been explored in the past [7, 19, 51], it was rarely able to produce efficient code. In fact, optimizations for CPUs and even generic compiler transforms, such as common sub-expression elimination or loop-invariant code motion, are hindered by the lack of analyzable representations of parallel constructs inside the compiler [34]. As representations of parallelism within a mainstream compiler have only recently begun to be explored [8, 10, 28, 43, 48], existing transformations are limited and tend to apply to simple CPU codes only.

We propose a compiler model for most common GPU constructs: multi-level parallelism, level-wide synchronization, and level-local memory. In contrast to source and AST-level approaches, which operate before the optimization pipeline, and existing compiler approaches, which model synchronization as a black-box optimization barrier, we model synchronization from memory semantics. This allows synchronization-based code to interoperate with existing optimizations and enables novel parallel-specific optimizations.

Our model is implemented using MLIR [30] and LLVM [29] and leverages MLIR’s nested-module approach for GPU [18]. We extended the Polygeist [35] C/C++ frontend to support CUDA and to produce MLIR which preserves high-level parallelism structure. Our prototype compiler is capable of compiling PyTorch CUDA kernels, as well as other compute-intensive benchmarks, to any CPU architecture supported by LLVM. In addition to transformations accounting for the differences in the execution model, we also exploit parallelism on the CPU via OpenMP. Finally, our MocCUDA PyTorch integration allows us to compile and execute CUDA kernels in absence of a GPU while substituting unsupported calls.

The correctness and efficiency of our end-to-end translation is evaluated by compiling Rodinia CUDA benchmarks [4]

as well as PyTorch CUDA kernels. When targeting a commodity CPU, our OpenMP-accelerated CUDA code yields comparable performance with the reference OpenMP implementations from the Rodinia suite, as well as improved scalability. When using our framework to run PyTorch on the CPU-only Fugaku Supercomputer, we achieve roughly twice the images processed per second by the conv2d kernel from Resnet-50 [21] compared to the OneDNN-based PyTorch CPU backend, and comparable performance to the hand-tuned overall training.

Overall, our paper makes the following contributions:

- A common high-level and platform-agnostic representation of SIMD-style parallelism backed by a semantic definition of barrier synchronization that ensures correctness through memory semantics, and thus transparent application of existing optimizations.
- Novel parallel-specific optimizations which can exploit our high-level parallel semantics to optimize programs.
- An extension to the Polygeist C/C++ MLIR frontend capable of directly mapping GPU and CPU parallel constructs into our high-level parallelism primitives.
- An end-to-end transpilation of CUDA to CPU for a subset of the Rodinia [4] benchmark suite and the internal CUDA kernels within PyTorch [38] necessary to run Resnet-50 on the CPU-only Fugaku supercomputer.

2 Background

Mainstream compilers like Clang and GCC lack a unified high-level representation of parallelism. Compiling parallel constructs in frameworks like CUDA, OpenMP, or SYCL, forces the body of a parallel region to exist within a separate (closure) function which is invoked by a parallel runtime. Concepts such as thread index or synchronization are then represented separately, often through opaque intrinsic calls. As the compiler historically lacked information about parallelism and effects of the involved runtimes, any parallel construct also inadvertently acted as a barrier to optimization. While there have been attempts [8, 10, 28, 34, 43, 48, 54] in recent years to improve representations for CPU parallel constructs, accelerator programming comes with additional challenges. The unique programming model and complex memory hierarchy have left high-level representations of GPU parallelism within mainstream compilers under-explored.

2.1 GPU Compilation

Consider the CUDA program in Fig. 1, which normalizes a vector. When compiled using Clang, the GPU program is a separate compilation unit. This prevents any optimization between the GPU kernel and the CPU calling code. In the case of Fig. 1, the total work of the program in a traditional compiler is $O(N^2)$, due to the $O(N)$ call to sum being performed for each thread. However, if the call to sum is performed only once prior to the kernel call, e.g., by performing

```
__device__ float sum(float* data, int n) { ... }
__global__
void normalize(float *out, float* in, int n) {
    int tid = blockIdx.x + blockDim.x * threadIdx.x;
    // Optimization: Compute the sum once per block.
    // __shared__ int val;
    // if (threadIdx.x == 0) val = sum(in, n);
    // __syncthreads();
    float val = sum(in, n);
    if (tid < n)
        out[tid] = in[tid] / val;
}
void launch(int *d_out, int* d_in, int n) {
    normalize<<<(n+31)/32, 32>>>(d_out, d_in, n);
}
```

Figure 1. A sample CUDA program normalize, which normalizes a vector and the CPU function launch launching the kernel. Each GPU threads calls sum, resulting in $O(N^2)$ work. Using shared memory (commented) reduces the work to $O(N^2/B)$ at extra resource cost. Computing sum before the kernel reduces work to $O(N)$.

loop-invariant code motion (LICM), the work would reduce to $O(N)$. A less effective variant of this optimization could reduce the work to $O(\frac{N^2}{B})$ through the use of shared memory. MLIR provides a nested-module representation for GPU programs that supports host/device code motion [18], but parallel code motion has not been implemented. In GPU to CPU code motion, LICM out of a parallel loop is always legal as any former device memory is also available on the host.

2.2 MLIR Infrastructure

MLIR is a recent compiler infrastructure designed for reuse and extensibility [30]. Rather than providing a predefined set of instructions and types, MLIR operates on collections of *dialects* that contain sets of interoperable user-defined operations, attributes and types. Operations are a generalization of IR instructions that can be arbitrarily complex, in particular, contain regions with more IR thus creating a nested representation. Operations define and use values that obey single static assignment (SSA) [6]. For example, MLIR dialects may model entire physical or virtual instruction sets such as NVVM (virtual IR for Nvidia GPUs), other IRs such as LLVM IR [29], higher-level control flow constructs such as affine loops, parallel programming models such as OpenMP and OpenACC, machine learning graphs, etc.

MLIR supports GPU thanks to the eponymous dialect, which defines the high-level SIMD programming model, host/device communication, and a set of platform-specific dialects: NVVM (CUDA), ROCm (ROCD) and SPIR-V. MLIR’s approach to GPU programming benefits from a *unified* code representation. Since an MLIR module may contain other modules, the “host” translation unit may embed the “device” translation unit as IR rather than file reference or binary blob. This approach provides host/device optimization opportunities unavailable to other compilers, in particular to move code between host and device [18].

```

221 // Kernel launch is available within the calling
222 // function, enabling optimizations across the
223 // GPU/CPU boundary.
224 func @launch(%h_out : memref<?xf32>,
225             %h_in : memref<?xf32>, %n : i64) {
226     // Parallel for across all blocks in a grid.
227     parallel.for (%gx, %gy, %gz) = (0, 0, 0)
228         to (grid.x, grid.y, grid.z) {
229             // Shared memory = stack allocation in a block.
230             %shared_val = memrefalloca : memref<f32>
231             // Parallel for across all threads in a block.
232             parallel.for (%tx, %ty, %tz) = (0, 0, 0)
233                 to (blk.x, blk.y, blk.z) {
234                 // Control-flow is directly preserved.
235                 if %tx == 0 {
236                     %sum = func.call @sum(%d_in, %n)
237                     memref.store %sum, %shared_val[] : memref<f32>
238                 }
239                 // Synchronization via explicit operation.
240                 polygeist.barrier(%tx, %ty, %tz)
241                 %tid = %gx + grid.x * %tx
242                 if %tid < %n {
243                     %res = ...
244                     store %res, %d_out[%tid] : memref<?xf32>
245                 }
246             }
247         }
248     }
249 }
```

Figure 2. Polygeist/MLIR representation of the shared-memory version of the CUDA launch/normalize code from Fig. 1. The kernel call is made available directly in the host code which calls it. The parallelism is made explicit with parallel for loops across the blocks and threads. Shared memory is placed within the block parallel for, allowing access from any thread in the same block, but not a different block.

2.3 Polygeist

Polygeist is a C/ C++ frontend for MLIR based on Clang [35]. It is capable of translating a broad range of C++ programs into a mix of MLIR dialects that preserve elements of the high-level structure of the program. Specifically, Polygeist preserves structured control flow (loops and conditionals) as MLIR SCF dialect operations and simplifies analyses by preserving multi-dimensional array constructs whenever possible by relying on the MLIR’s multi-dimensional memory reference (memref) type. Finally, Polygeist is able to identify parts of the program suitable for polyhedral optimization [14] and represent them using the Affine dialect.

3 Approach

We extended the Polygeist compiler [35] to directly emit parallel MLIR from CUDA. This leverages the unified CPU/GPU representation to allow the optimizer to understand host/device execution, and to enable optimization across kernel boundary. The use of existing MLIR’s first-class parallel constructs (`scf.parallel`, `affine.parallel`) enables us to target existing CPU and GPU backends. Finally, MLIR’s extensible operation set allows us to define custom instructions, with relevant properties and custom optimizations.

| | |
|---|--|
| <pre> <code>--global__ f() {</code> <code> codeA();</code> <code> barrier();</code> <code> codeB();</code> <code>}</code> </pre> | <pre> <code>--global__ f() {</code> <code> // 0<=t.x< blockDim.x</code> <code> A[threadIdx.x] = ...; // W A[i]: i==t.x</code> <code> barrier();</code> <code> // RW A[i]: i!=t.x</code> <code> ... = A[threadIdx.x]; // R A[i]: i==t.x</code> <code>}</code> </pre> |
|---|--|

Figure 3. Left: A program containing a barrier between two arbitrary instructions. **Right:** Barrier semantics can be refined memory addresses accessed by operations above/below it in all threads *except* the current one.

We define the representation of a GPU kernel launch as follows (illustrated in Fig. 2):

- A 3D parallel for-loop over all blocks in the grid.
- A stack allocation for any shared memory, scoped to be unique per block.
- A 3D parallel for-loop over all threads in a block.
- A custom Polygeist barrier operation that provides equivalent semantics to a CUDA synchronization.

This procedure enables us to represent any GPU program in a form that preserves the desired semantics. It is fully understood by the compiler and is thus amenable to compiler optimization. Moreover, by representing GPU programs with general parallelism, allocation, and synchronization constructs, we are not only able to optimize the original program, but also retarget it for a different architecture.

3.1 Barrier Semantics

A CUDA `__syncthreads` function guarantees that all threads in a block have finished executing all instructions prior to the function call, before any threads executes any instruction after the call. Traditionally, compilers represent such functions as opaque optimization barriers that could touch all memory, and forbid any transformation involving them.

In our system, we chose to represent thread-level synchronization through a new `polygeist.barrier` operation. Unlike other approaches, `polygeist.barrier` (hence referred to as simply `barrier`) aims to only prevent transformations that would change externally visible behavior. Rather than disallowing any code motion across a barrier, we can successfully achieve the desired semantics by defining `barrier` to have specific memory properties, represented as a collection of memory locations (including unknown), and memory effect type (read, write, allocate, free), as is standard within MLIR. Consider the simple program in Fig. 3(left). The impact of the synchronization can only be observed if `codeA` and `codeB` access the same memory. Moreover, if both only read the same memory location, the synchronization is also unnecessary. We can enumerate the remaining cases:

- (1) `codeA` writes, `codeB` loads
- (2) `codeA` loads, `codeB` writes
- (3) `codeA` writes, `codeB` writes

The barrier having the write behavior of `codeA` would ensure correctness of (1): the load in `codeB` could not be

```

331
332 parallel %i = 0 to 10 {
333     %x = load data[%i]
334     %y = load data[2 * %i]
335     %a = fmul %x, %x
336     %b = fmul %y, %y
337     %c = fsub %x, y
338     barrier
339     call @use(%a, %b, %c)
340     ...
341 }
342
343 %x_cache = memref<10xf32>
344 %y_cache = memref<10xf32>
345 parallel %i = 0 to 10 {
346     %x = load data[%i]
347     %y = load data[2 * %i]
348     store %x, %x_cache[%i]
349     store %y, %y_cache[%i]
350 }
351 parallel %i = 0 to 10 {
352     %x = load %x_cache[%i]
353     %y = load %y_cache[%i]
354     %a = fmul %x, %y
355     %b = fsub %y, %z
356     call @use(%a, %b)
357     ...
358 }

```

Figure 4. Parallel loop splitting around a barrier: the code above the barrier is placed in a separate parallel “for” loop from the code following the barrier. This transformation eliminates the barrier, while preserving the semantics. The min-cut algorithm stores $\%x$ and $\%y$, which are then used to recompute $\%a$, $\%b$, and $\%c$ in the second loop.

352 hoisted above the barrier, as it would appear to read a dif-
353 ferent value. Symmetrically, the barrier having the write
354 behavior of codeB ensures the correctness of (2). Thus, the
355 union of the writing behaviors of codeA and codeB is suffi-
356 cient to prevent illegal movement of loads across the barrier.

However, this does not prevent writes from being moved. For example, codeB could be duplicated above the barrier in (3), and it would appear to have the same final memory state since the extraneous write before the barrier would never be read. Thus, we also define the barrier to have the reading behavior of codeA and codeB.

This model can be extended to include memory effects of all operations in the parallel loop which may have been executed before, or after, a given barrier. On a control flow graph with explicit branches, this requires exploring the operations within predecessors or successors, respectively. However, operating on MLIR's structured control flow level, with explicit operations for loops and conditionals, simplifies the analysis. Furthermore, if more than one barrier is present in the same block, it is unnecessary to look past it.

Given a sufficiently expressive side effect model, the memory semantics of the barrier can be further expanded. While barriers enforce ordering reads/writes to the same location from *different* threads, the natural execution order is sufficient within one thread. Therefore, barriers need not capture the memory effects of operations where the address is an *injective function* of the thread identifier. Raising memory accesses into affine forms, when possible, enables precise analysis. Consider the code in Fig. 3(right). The read and write expressions around the barrier have the affine access sets $\mathcal{A}_o = \{A(i) : i = tx\}$ where tx is the thread \times identifier. The barrier has the affine access set $\mathcal{A}_b = \{A(i) : i \neq tx\}$. Since the sets of accessed addresses do not overlap, $\mathcal{A}_o \cap \mathcal{A}_b = \emptyset$,

```

parallel for %id=0 to N {
    for %j = 5 to 0 {
        if (%id < 2^%j)
            A[%id] += \
                A[%id + 2^%j]
        barrier
    }
}

for %j = 5 to 0 {
    parallel for %id=0 to N {
        if (%id < 2^%j)
            A[%id]=A[%id + 2^%j]
        barrier
    }
}

```

Figure 5. Left: A shared memory addition, which consists of a kernel call which contains for loop with a barrier inside. **Right:** The same code with the barrier now directly within the parallel loop by performing an interchange of the parallel for loop with the serial for loop.

code motion across the barrier is allowed. Conceptually, the write to $A[\text{threadIdx.x}]$ always happens before the read within the same thread so the barrier is unnecessary. In contrast, if the load and store to A were offset by 1, the barrier would be necessary as the data loaded after the barrier would be stored by a different thread. Aliasing guarantees must be checked when more than one base address is involved.

3.2 Barrier Lowering

To enable GPU programs to run on a CPU, we must efficiently emulate the synchronization behavior of GPU programs. Whereas the memory semantics in Section 3.1 enable us to preserve the correctness of barriers during optimization, this section discusses how to implement the barrier on a CPU.

CPU architectures have no notion of thread blocks, nor the barrier instruction which waits on this conceptual grouping of threads. Instead, we use regular CPU threads and work sharing to distribute the thread-block loop iterations across them. Conceptually, this differs from the GPU execution model in which threads execute one iteration each. Work sharing requires each thread to execute multiple iterations sequentially, making it impossible to synchronize in the middle of iterations, but only at the end of the loop.

To address this, we developed a new barrier elimination technique for our MLIR representation. As discussed in Section 7, several approaches have been explored in the past including loop fission and continuation passing. Our approach is an extension of the former combining two transformations: *parallel loop splitting* and *parallel loop interchange*.

3.2.1 Parallel Loop Splitting. Suppose a barrier has the kernel function (or, in our representation, parallel for loop) as its direct parent. It can be eliminated by splitting the loop around the barrier into two parallel for loops that run the code before and after the barrier, respectively. If the code before the barrier created SSA values that were used after it, these must be either stored or recomputed in the second parallel loop. We use the technique similar to one in [36] to determine the minimum amount of data that needs to be stored. Specifically, we create a graph of all SSA values. We then mark each value definition that cannot be recomputed

```

441           %helper = alloca memref<i1>
442   parallel for %i=0 to N {
443     scf.do {
444       parallel for %i=0 to N {
445         do {
446           run(%i)
447           barrier
448         } while(condition())
449       }
450     }
451   }
452 }
```

Figure 6. Parallel interchange around a `while` loop. As the `condition()` function call must be executed on each thread to preserve correctness, a helper variable is used which holds the value of the call on the first thread.

(e.g. loads from overwritten memory) before the barrier as source, and values used after the barrier as sinks. We derive the minimum amount of data needing to be stored by performing a minimum branch cut on this graph.

3.2.2 Parallel Loop Interchange. Not all barrier operations have a parallel for as their immediate parent, some may be nested in other control flow operations. We created a model that specifies what instructions may run in parallel. With the sole exception of `barrier`, our representation does not require any specific ordering or concurrency to the program. Therefore it is legal (though potentially a reduction in parallelism) to add additional barriers. We can use this property to implement barrier lowering for control flow.

Consider a control-flow construct C containing a barrier and nested in a parallel for. Adding barriers immediately around C will result in parallel loop splitting directly above and below C. As a result, the operations above and below C will be separated into their own parallel for and C will be the sole operation in the middle loop. We can then apply one of the following techniques to interchange C with the parallel for, thus making the `barrier`'s parent a parallel for.

Consider the case of a serial for loop containing a barrier, Fig. 5. This pattern is common in GPU code, e.g., to implement a reduction across threads [20]. As `barrier` must wait for all threads, each thread must execute the same number of barriers. Therefore, the number of iterations of the inner loop is the same for all threads, allowing for loop interchange.

While an `if` statement can be considered a loop with zero or one iteration, directly interchanging it with the surrounding parallel for when necessary is more efficient.

Whereas for loops in MLIR have their iteration count specified prior to execution, a `while` loop supports a dynamic exit condition, like in Fig. 6. Since correctness requires executing `condition()` in every thread, a direct interchange would not be legal. However, GPU synchronization semantics require the iteration count to be the same in all threads. Therefore, one can still perform an interchange using a helper variable to store the result of the condition.

```

496 __global__ void bpnn_layerforward(...) {
497   __shared__ float node[HEIGHT];
498   __shared__ float weights[HEIGHT][WIDTH];
499   if ( tx == 0 ) node[ty] = input[index_in] ;
500   // Unnecessary Barrier #1
501   __syncthreads();
502   // Unnecessary Store #1
503   weights[ty][tx] = hidden[index];
504   __syncthreads();
505
506   for ( int i = 1 ; i <= log2(HEIGHT) ; i++ ){
507     if( ty % pow(2, i) == 0 )
508       weights[ty][tx] += weights[ty+pow(2, i-1)][tx];
509     __syncthreads();
510
511     hidden[index] = weights[ty][tx];
512     // Unnecessary Barrier #2
513     __syncthreads();
514
515     if ( tx == 0 ) out[by * hid + ty] = weights[tx][ty];
516   }
517 }
```

Figure 7. An example CUDA kernel from the Rodinia back-prop test that contains unnecessary synchronization and unnecessary use of shared memory.

This illustrates one of the advantages of building off of MLIR/Polygeist. By preserving high level program structures, we can use more efficient patterns to remove barriers.

4 Parallel Optimization

The high-level representation of both parallelism and GPU programs provided by Polygeist/MLIR enables a variety of optimizations. These include general optimizations that would apply to any parallel program as well as specific optimizations in the context of GPU to CPU conversion.

4.1 Barrier Elimination & Motion

As GPU-style barriers have to be specially transformed to support CPU architectures, eliminating or simplifying any barriers can have dramatic effects. Moreover, even when running GPU code on the GPU, barrier elimination is highly useful as any synchronization reduces parallelism. Much of the infrastructure for barrier elimination/simplification comes directly from its memory behavior defined in Section 3.1. Given a barrier B, let M_{before}^\dagger be the union of memory effects before B until either another barrier or the start of the parallel region, and let the union of memory effects after B until the end of the parallel region M_{after} . If there are no memory effects to the same location across the barrier other than a read-after-read (RAR) (i.e. $(M_{before}^\dagger \cap M_{after}) \setminus \text{RAR} = \emptyset$), the barrier has its behavior subsumed by the prior barrier and can be eliminated. The symmetric condition $M_{before} \cap M_{after}^\dagger \setminus \text{RAR} = \emptyset$ indicates that the barrier is subsumed by a subsequent barrier. A specific case of a removable barrier is one that has no memory effects at all.

551 For example, consider the code in Fig. 7, which comes
 552 from the backprop Rodinia benchmark [4]. The first and
 553 last `__syncthreads` instructions are unnecessary. This can
 554 be proven from our memory-based barrier elimination algo-
 555 rithm above as follows. For the first barrier, M_{before} (going all
 556 the way to the start) contains only a write to node and a read
 557 from input. M_{after}^\dagger (going to the second `__syncthreads`)
 558 contains a write to weights and a read from hidden. None
 559 of these conflict if, given the calling context, the pointers are
 560 known not to alias. Thus, it is safe to eliminate the barrier.
 561

562 The same memory analysis can also be applied to perform
 563 barrier motion. One simply needs to place a fictitious barrier
 564 at the intended location and check if the previous memory
 565 analysis would deduce that the current barrier is unnecessary,
 566 thereby permitting barrier motion.

4.2 Memory-to-register promotion across barriers

567 One of the goals of defining barrier's semantics from its
 568 memory behavior is to enable memory optimizations to op-
 569 erate correctly and effectively in code that contains barriers.
 570 As described in Section 3.1, barriers have the memory be-
 571 havior of the code above and below them with the notable
 572 exception of an access from the current thread. This hole
 573 is important as it enables memory-to-register promotion
 574 (`mem2reg`) to operate on thread-local memory such as local
 575 variables. This optimization can replace slow memory reads
 576 with fast registers. For example, consider again the code in
 577 Fig. 7. Consider the load and store to `weights[ty][tx]` la-
 578 beled "Unnecessary Store #1" and "Unnecessary Load #1",
 579 and the sync in between the two. The only value that can be
 580 loaded at that point is the same value which was stored ear-
 581 lier, a register containing the value loaded from `hidden`. As
 582 that same location is overwritten before anyone else could
 583 read from `weights`, the first store also can be safely eli-
 584 minated once the load is removed. During `mem2reg`, Polygeist
 585 can derive this forwarding property, since the hole in the
 586 memory properties described in Section 3.1 allows it to de-
 587 duce that the barrier operation does not overwrite the store
 588 for the current thread. As a result, traditional load and store
 589 forwarding correctly operates on the barrier code.
 590

4.3 Parallel loop-invariant code motion

591 The traditional loop-invariant code motion optimization
 592 aims to move an instruction `I` outside serial "for" loops, re-
 593 ducing the number of times `I` is executed. If `I` may access
 594 memory, or has other side effects, in addition to checking
 595 that the operands of `I` are themselves loop invariant, the
 596 compiler must check that no other code within the "for" loop
 597 conflicts with the memory access performed by `I`.
 598

599 On present compilers, while it is possible to apply loop-
 600 invariant code motion to serial for loops within GPU kernels,
 601 it is not possible to apply loop-invariant code motion to
 602 hoist instructions outside of a kernel call. This is in part due
 603 to

```
591 omp.parallel {
 592   omp.wsloop %i= 1 to 10 {
 593     codeA(%i)
 594   }
 595 }
 596 omp.parallel {
 597   omp.wsloop %i= 1 to 10 {
 598     codeA(%i)
 599   }
 600 }
```

```
606   omp.parallel {
 607     omp.wsloop %i=1 to 10 {
 608       codeA(%i)
 609     }
 610   }
 611 }
```

Figure 8. Example of OpenMP parallel region fusion. Fuse
 614 two adjacent OpenMP parallel regions by inserting a barrier
 615 to allow the threads to be initialized once instead of twice.
 616

```
617 for (i=0; i<N; i++) {
 618   #pragma omp parallel for
 619   for (j=0; j<10; j++) {
 620     body(i, j);
 621   }
 622 }
```

```
623 #pragma omp barrier
 624 }
```

Figure 9. Example of OpenMP parallel region hoisting. This
 624 can be seen as an extension of parallel region fusion across
 625 "regions" corresponding to each iteration of the outer loop.
 626

627 to the fact that GPU kernels are kept in a separate module
 628 from the CPU code which calls them, as well as a lack of
 629 understanding of parallelism (see Fig. 1).
 630

631 Counter-intuitively, with the right semantics we can apply
 632 loop-invariant code motion to parallel for loops even if we
 633 would not be able to apply it to an equivalent serial loop. We
 634 will rely on the fact that semantics of our program permits
 635 us to arbitrarily interleave iterations of a parallel "for" loop
 636 as long as we maintain the orderings required by barriers.
 637 As such, it is legal, though not necessarily fast, to run the
 638 program in lock-step. In other words, if a parallel for loop
 639 had 10 instructions, each thread can execute instruction 1
 640 before any thread executed instruction 2, and so on. As a
 641 consequence, it is now legal to hoist an instruction so long
 642 as its operands are invariant and no *prior* instruction in the
 643 parallel for loop conflicts with `I`. In other words, one does not
 644 need to check if `I` conflicts with any subsequent instruction
 645 in the parallel for loop to enable hoisting.
 646

4.4 Block Parallelism Optimizations

647 OpenMP is our primary target for parallel execution on the
 648 CPU. It implements parallel "for" loops as two constructs.
 649 First, the loop is outlined into a function which is called
 650 once per thread, representing OpenMP's "parallel" construct.
 651 Then, within the outlined function, the iteration space is dis-
 652 tributed across threads, representing OpenMP's "workshar-
 653 ing loop" construct. OpenMP also has a "barrier" construct,
 654 but with semantics *different* than that of a GPU barrier.
 655

656 When multiple parallel loops are executed in a row, e.g.,
 657 following the barrier lowering from Section 3.2, the overhead
 658 of thread management can be reduced by fusing adjacent
 659

OpenMP "parallel" constructs [9] *without* fusing the work-sharing loops (see Fig. 8), thus not undoing the barrier lowering. This can be extended to moving the OpenMP parallel region outside the surrounding "for" in Fig. 9, initializing threads once rather than N times. Applying these to control flow constructs enables all of the "for" loops generated by performing parallel loop fission on a block to have their OpenMP "parallel" (but not work sharing loops) fused.

As GPU programs tend to be written with high parallelism in mind, the parallelism provided by the different blocks may already saturate the number of available cores alone. If there is no use of shared memory, the block and thread parallelism can be collapsed into a single OpenMP parallel for, which will evenly divide the total iteration space in a single parallel region. However, if there is shared memory, our tool will generate nested parallel regions to represent the shared memory allocation. In this case, the additional overhead from the nested OpenMP parallel regions may outweigh the potential added parallelism. In addition, parallelizing the inner loops may lead to adverse memory effects such as false sharing, further penalizing performance [55, 57]. As such, we also support an optimization for serializing any nested OpenMP parallel regions. Performing such serialization may leverage memory locality to improve performance.

5 MocCUDA: Integration into PyTorch

One of our goals is to support execution of originally GPU codes on a CPU-only supercomputer such as Fugaku [42]. We focus on PyTorch [38] that has not been ported to the A64FX architecture and therefore uses naive fallback CPU kernels. Observing that CPUs with high-bandwidth memory are likely to benefit from GPU-style optimization, we implement MocCUDA, a mock GPU backend for PyTorch that redirects the calls to CUDA runtime and libraries to our implementations or A64FX-specific math libraries [17]. We collect statistics of library calls and may optionally substitute them with CPU versions transpiled by Polygeist.

6 Evaluation

We demonstrate the advantages and applicability of our approach on two well-known GPU benchmark suites: a subset of the GPU Rodinia benchmark suite [4] and a PyTorch implementation of a Resnet-50 neural network. These benchmarks were chosen to 1) provide a rough performance comparison of our GPU to CPU compilation on a benchmark suite (Rodinia) that has hand-coded CPU versions and 2) demonstrate a successful end-to-end integration of our system into a useful and real application (PyTorch Resnet-50) on Supercomputer Fugaku, which does not have any GPUs. Additionally, we compare the performance of our approach to the existing MCUDA [51] tool on a CUDA matrix multiplication.

For Rodinia, we compare our translated CUDA to CPU code against OpenMP versions of the benchmarks, where

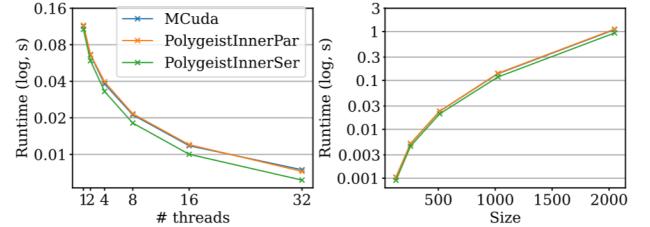


Figure 10. PolygeistInnerPar performs similarly to MCUDA; PolygeistInnerSer outperforms MCUDA. PolygeistInnerSer disables inner loop parallelization similarly to MCUDA, whereas PolygeistInnerPar keeps both the blocks and threads parallel. Left: Average runtime as a function of thread count (averaging over matrix sizes). Right: Average runtime as a function of matrix size (averaging over thread counts).

they exist, as well as a run on a GPU. For the PyTorch Resnet-50, we compare against the "native" and oneDNN backends.

Polygeist¹ was compiled using LLVM 15 (git 00a1258). For the PyTorch Resnet-50, we compile Pytorch v1.4.0 using Nvidia's CUDA 11.6 SDK for Arm², LLVM 13, and Fujitsu's SSL2 v1.2.34 library. For the baseline PyTorch measurements, we use Fujitsu's pre-installed PyTorch (v1.5.0).

We evaluate the Rodinia and matrix multiplication tests on an AWS c6i.metal instance (dual-socket Intel Xeon Platinum 8375C CPU at 2.9 GHz with 32 cores each and 256 GB RAM) running Ubuntu 20.04. Measurements were performed on the first socket, with hyperthreading and turbo boost disabled. Each number is the median of at least 5 repetitions.

6.1 Comparison to MCUDA

First, we compare with the previous work in MCUDA [51]. MCUDA is an AST-level tool which produces new CPU C/C++ as an output and uses loop fission to handle synchronization. As a source-to-source tool, MCUDA only handles a fraction of the input language, making it unable to run on Rodinia programs. Instead, we compare the runtimes of a matrix multiplication kernel across a range of threads (1–24) and matrix sizes (128×128 – 2048×2048) in Fig. 10. Polygeist with all optimization excluding serialization of the inner loop (PolygeistInnerPar) produces code within 1.3% of MCUDA on average. PolygeistInnerPar has a 1.5% slowdown on 1 thread, and 3.2% speedup on 32 threads. This behavior is caused by OpenMP overhead in handling nested parallel constructs. In fact, MCUDA only parallelizes the outermost loop. When Polygeist also serializes the inner loops (PolygeistInnerSer), it achieves an overall 14.9% speedup over MCUDA, with a 4.5% speedup on 1 thread and 21.7% speedup on 32 threads.

¹MocCUDA and Polygeist are available at tinyurl.com/thvyhndx and tinyurl.com/mvkpysvm.

²Even though we will run PyTorch on a GPU-less system, we must compile PyTorch on a CUDA-enabled system to ensure the correct code is emitted. We also prevented inlining of three Pytorch functions.

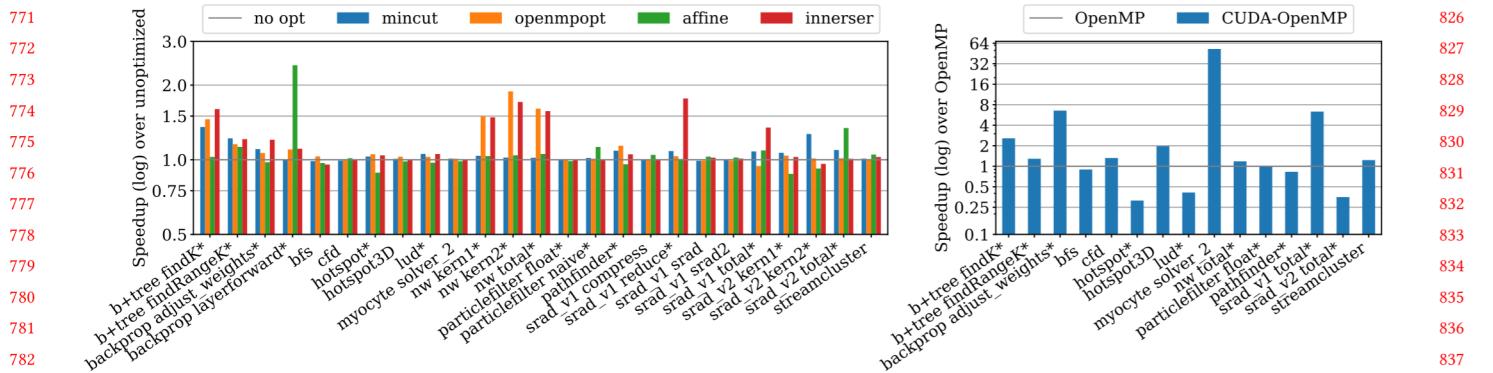


Figure 11. Left: Relative speedup (higher is better) applying various parallel optimizations. Right: Speedup of transcompiled CUDA-to-OpenMP compared against native OpenMP code (when available). Asterisks denote barriers within the benchmark.

6.2 Use case 1: Rodinia Benchmarks

We benchmarked the 14 benchmarks that are currently supported by Polygeist, and had a nontrivial runtime.³ We verified correctness by comparing the program outputs produced by compiling with nvcc and executed on a GPU, and compiled by our flow and executed on a CPU. We also employed the use of CPU-based parallel and undefined behavior analysis tools, which via our tool, allowed us to successfully diagnose and repair one race bug and several undefined memory bugs in the original CUDA code. We inserted timing measurements across kernels and/or computational portions of the code that include kernels, in some cases multiple per benchmark. Where possible, we time equivalent portions of the OpenMP versions of the same benchmarks.

We compare the Rodinia CUDA benchmarks compiled for the CPU with the Rodinia OpenMP versions of the benchmark in Fig. 11(right). While there is some variation from benchmark to benchmark, overall our approach is on par with the hand-coded versions of the benchmarks, and even nets a 58% geomean performance improvement, when the inner serialization optimization is enabled. Without inner serialization, we still see a geomean speedup of 34%. The speedup for myocte is largely due to fewer instruction and data cache misses on the transcompiled code, which comes from optimizations which specialize the (parallel) to kernel call context, as well as the CUDA version employing fewer branches. The speedup for backprop is partially due to parallel optimizations (see Fig. 11(left)) and partially due to the CUDA code being implemented with a linear array, as required by CUDA, instead of the double-pointer used in the OpenMP code. The srad_v1 benchmark benefits from a shared memory reduction in addition to parallel optimizations which eliminate most barriers and shared memory. In

³The hybridsort, kmeans, leukocyte, mummergpu huffman and heartwall use unsupported C++ or CUDA features within Polygeist (virtual functions and texture memory). The lavaMD and dwt2d benchmarks use ill-formed C++ with undefined behavior due to reading from uninitialized memory. The nn and gaussian tests ran in ≤ 0.005 seconds.

contrast, hotspot and pathfinder see a slowdown compared against native OpenMP code, due to duplicated computation in order to reduce synchronization and make better use of plentiful GPU parallelism. The slowdown for the transpiled CUDA version of lud is due to being written with a transposed loop ordering in contrast to the OpenMP code.

We test the scaling properties of our approach by comparing transpiled CUDA with native OpenMP kernels in Fig. 12. Transpiled CUDA codes generally scale much better than the native OpenMP versions. As most CUDA programs are written with thousands of threads in mind, this indicates that our framework was able to preserve that parallelism as the GPU-specific constructs were being rewritten for CPU-compatible equivalents. On 32 threads without inner serialization, transpiled CUDA codes had a geomean speedup of 16.1 \times across all tests. As OpenMP versions of benchmarks do not exist for all tests, if we consider only CUDA codes for which there exists an OpenMP version, we find a geomean speedup of 14.0 \times , whereas OpenMP has only a speedup of 7.1 \times . Serializing the inner loop slightly reduces scalability, but still results in improved scalability over OpenMP, finding a geomean speedup of 14.9 \times over all tests with inner serialization enabled, and a 12.5 \times speedup on codes with OpenMP versions.

We perform an ablation analysis to study how individual optimizations impact performance. The “mincut” series in Fig. 11(left) shows performance measurements for our approach with the optimization outlined in Section 3.2.1 to reduce the amount of data preserved across barriers. This is only relevant for benchmarks containing barriers (marked by an asterisk in the Figure). On applicable benchmarks, mincut provides a 5.8% geomean speedup. The “openmpopt” series in Fig. 11(left) demonstrates the impact of OpenMP region merging and similar optimizations and results in a 10.5% geomean speedup. The “affine” series in Fig. 11(left) shows the result of raising control flow to their affine variants and enabling simple serial and parallel loop optimizations (such as loop unrolling and re-indexing). While this produces a geomean speedup of 5.4% across the board, it results in a

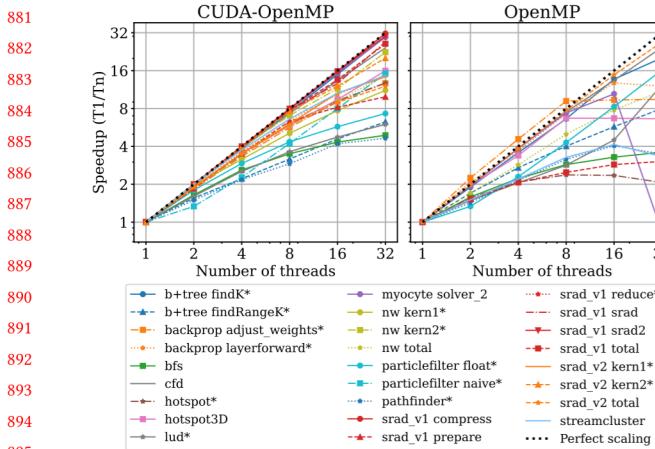


Figure 12. Scaling behavior behavior of CUDA Rodinia kernels, when run on the CPU with OpenMP, and OpenMP Rodinia kernels (where available), using 32 threads. Not all Rodinia CUDA kernels have OpenMP versions.

2.4× speedup for the backprop layerforward test as it results in a loop containing synchronization being fully unrolled and reduced to if statements.

6.3 Use case 2: Pytorch/Resnet50 Test

To evaluate the PyTorch Resnet-50, we execute a full node-parallel training run on one TofuD unit of the Fugaku FX1000 supercomputer, comparing against the native PyTorch CPU backend and the optimized oneDNN backend, as available. We replaced the functions related to computing log-likelihood with Polygeist-transpiled functions as their CUDA kernels use barriers and their CPU versions contain naive implementations, and dispatched other calls to relevant libraries.

We ran multiple forward and back propagation passes of Resnet-50 on 224×224 ImageNet in a data-parallel fashion. We employ Horovod’s synthetic benchmarking script [45]. We build Horovod v0.19.5 with CUDA, LLVM, and Fujitsu’s MPI library to enable multi-node, distributed deep learning on top of Pytorch. We assign one MPI rank per A64FX core memory group (CMG), emulating up to 4 GPUs per node, and scale the test from one node (2 ranks) to 12 nodes (48 ranks) in one TofuD unit (smallest 2×3×2 torus) while keeping the number of OpenMP threads fixed at 12 to accommodate one thread per core. We use Pytorch v1.4.0 for our approach, while the other backends depend on Pytorch v1.5.0. Performance measurements were taken using Benchmarker [11], which sets up the neural network and test data, executes the layer, and returns the throughput. We run with batch sizes 1–12 on 1–64 threads, averaging across epochs.

MocCUDA systematically outperforms Fujitsu’s tuned oneDNN across batch sizes and thread counts, yielding up to 4.5× throughput increase (geomean 2.7×, min 1.2×) as shown in Fig. 13. MocCUDA with expert-written kernels is comparable to MocCUDA with Polygeist-generated kernels.

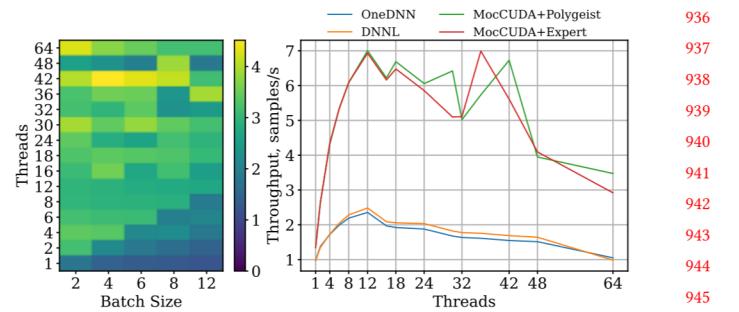


Figure 13. ResNet50 training on Fugaku node. Left: heatmap of relative throughput increase of “MocCUDA+Polygeist” over Fujitsu-tuned oneDNN, higher is better. Right: geomean throughput across batch sizes; “MocCUDA+Expert” uses an expert-written OpenMP kernel; “MocCUDA+Polygeist” uses the generated kernel.

The improvement can be explained by a combination of the PyTorch CPU design and performance characteristics of oneDNN. As Intel’s oneDNN [24] does not account for HBM available on A64FX, it uses cache-friendly direct convolutions instead of GEMM-based convolutions, less efficient in presence of HBM for Arm CPUs. While the custom fork of oneDNN tuned by Fujitsu [17], improves upon Intel oneDNN’s performance (though by a geomean of 6%), it still leaves room for performance improvements.

This demonstrates that our approach is capable of automatically deriving efficient versions of deep learning kernels (and potentially other applications) from their CUDA versions, thus addressing the limitations of missing or inefficient kernels for CPUs with high-bandwidth memory without the need for reverse or re-engineering the application.

7 Related Work

7.1 GPU to CPU Synchronization

One of the first tools for emulating GPUs on a CPU was provided directly by NVIDIA for debugging purposes and emulated each thread on the GPU with a distinct CPU thread. While functional, the large gap in the number of available threads makes the emulation inefficient.

MCUDA [51] (2008) performs an AST transformation of C GPU code to generate new C CPU code that calls a thread-independent parallel for routine. MCUDA pioneered the use of “deep fission” to handle synchronization, which splits parallel loops and other constructs at synchronization points in order to eliminate them. This fission technique is also applied in other tools: Ocelot [7] (2010), a binary-translation tool that parses PTX assembly into LLVM and just-in-time compiles kernel functions; POCL [25] (2015), a Clang/LLVM compiler pass for OpenCL; COX [19] (2021), another LLVM transformation pass for translation of CUDA that uses fission, and handles warp-level primitives; and even this work. While the intuition behind the fission approach is similar to that

used here, we apply fission inside of a high-level compiler, rather than either source or a low-level IR. As demonstrated in Section 3.1, performing fission on structured programs enables more efficient code transformations. While applying fission at a source-level misses the opportunity to run optimizations before fission (like barrier elimination) and applying fission at a low-level requires attempting to reconstruct the high-level structure, operating within MLIR allows us to both apply optimization and preserve high-level structure. Moreover, source-level tools tend to be quite fragile as they must re-implement parsing and semantics or the target language (e.g. C++), and as a result only operate on a limited subset of the input language, requiring re-engineering effort to replace unsupported constructs (like pointer arithmetic).

Another approach uses continuation-passing to handle synchronization by creating state machine of all synchronization points (e.g. “microthreading”) [50] (2010). Karrenberg and Hack [26] (2012) propose a continuation-passing approach in LLVM that includes an algorithm for detecting and reducing divergence in the control-flow-graph. Follow-up work minimizes live values to reduce memory traffic [32].

VGPU [39] (2021) is similar to NVidia’s virtual GPU, except using C++ thread and fence. Shared memory, implemented as a single global, is expanded by the number of blocks.

Prior work that operates at the low-level LLVM IR extends significant effort to reconstruct high-level constructs, such as loops and kernel configurations, required for either efficient fission or continuation passing. For example, POCL [25] runs canonicalizations and loop transformations to rewrite the control flow graph and attempt to recognize it as a specific form that can be handled. Prior work that operates at source/AST level (e.g. MCUDA), beyond still needing to recognize GPU-level concepts, cannot benefit from optimizations that simplify the code resulting in easier control flow.

In contrast, by operating on MLIR’s mix-of-abstractions, we are able to simultaneously preserve source-level structure and perform program transformations such as loop unrolling or LICM that can, e.g., remove nested synchronization.

7.2 Parallel Portability/IR, & OpenMP Optimizations

Several tools define new abstractions in the host language that are amenable to CPU or GPU execution. Examples include ISPC [40], RAJA [2], Kokkos [13], or MapCG [23] (limited to map-reduce code) in C++, Loo.py [27] in Python, and KernelAbstractions.jl [5] in Julia. These approaches provide performance portability for any new code written with them. However, any existing code must be rewritten in said framework and may not compose with other frameworks/languages.

Several pieces of prior art discuss parallel intermediate representations, such as Tapir [43] for representing Cilk [16] in LLVM; OpenMPIR [49] for representing OpenMP in LLVM, PPIR [44] for pattern trees, and the MLIR OpenMP Dialect; as well as SDF3 [52] for visually representing concurrency as a control-flow graph. These works primarily focus on the

representation for their particular style of parallelism (e.g. OpenMP tasks in OpenMPIR), which does not include GPU-style barriers, rather than on parallel *transformations* (such as barrier elimination) or optimizations, with the exception of consistency/race checks or automatic parallelization [33, 37].

The use of OpenMP parallel region expansion is known to be beneficial [9]. Clang/LLVM optionally supports the transformation in a weaker form, namely merging of OpenMP parallel regions in the same control level [31].

7.3 Barriers

Several pieces of prior work explored the semantics of barrier or synchronization instructions, including in relation to GPUs. Work has been done to verify the correctness of barriers [1]. [47] experimentally evaluates the forward progress / fairness models of various GPU vendors. [46] implements a GPU barrier that applies across work-groups, as opposed to just within a work group. [53] add Java memory barriers to programs to ensure weak and sequential consistency semantics. They find that without synchronization and delay set analysis, introducing consistency semantics has an average 26.5× slowdown, whereas when using these analyses to insert fewer synchronizations can achieve a 10% and 26% slowdown for weak and sequential consistency, respectively.

8 Conclusion

By extending Polyeist/MLIR, we developed an end-to-end system capable of representing, optimizing, and transpiling CPU and GPU parallel programs. Being able to simultaneously represent and convert between distinct parallel frameworks is crucial as HPC increasingly relies on (heterogeneous) parallelism. A key component of our framework is the development of a high-level barrier operation, key to representing GPU programs, whose semantics can be fully defined by its memory behavior. Unlike prior representations of parallel barriers, our semantics enable direct integration of barriers within optimization. To validate the efficacy of our approach, we demonstrate GPU to CPU optimization and transpilation of a subset of the Rodinia benchmark suite on a commodity CPU and transcompile an efficient Resnet-50 from the PyTorch CUDA source to be run on the CPU-only Supercomputer Fugaku. While there is case-by-case variance due to implementation differences between CPU and GPU, the Rodinia benchmarks achieve a 58% geomean speedup of the transpiled GPU code over handwritten OpenMP versions. Similarly, we observe a $\approx 2\times$ speedup of CUDA PyTorch kernels above the native PyTorch CPU backend.

Currently, the transpiled GPU code keeps the same schedule when run on the CPU, except for the innermost loop serialization that improves performance. A fruitful avenue of future work may perform advanced rescheduling the code to better take advantage of CPU-style memory hierarchies.

1046
1047
1048
1049
1050
1051
1052
1053
1054
1055
1056
1057
1058
1059
1060
1061
1062
1063
1064
1065
1066
1067
1068
1069
1070
1071
1072
1073
1074
1075
1076
1077
1078
1079
1080
1081
1082
1083
1084
1085
1086
1087
1088
1089
1090
1091
1092
1093
1094
1095
1096
1097
1098
1099
1100

References

- [1] Alexander Aiken and David Gay. 1998. Barrier Inference. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Diego, California, USA) (*POPL '98*). Association for Computing Machinery, New York, NY, USA, 342–354. <https://doi.org/10.1145/268946.268974>
- [2] David Beckingsale, Richard Hornung, Tom Scogland, and Arturo Vargas. 2019. Performance Portable C++ Programming with RAJA. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming* (Washington, District of Columbia) (*PPoPP '19*). Association for Computing Machinery, New York, NY, USA, 455–456. <https://doi.org/10.1145/3293883.3302577>
- [3] H. Carter Edwards, Christian R. Trott, and Daniel Sunderland. 2014. Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *J. Parallel and Distrib. Comput.* 74, 12 (2014), 3202–3216. <https://doi.org/10.1016/j.jpdc.2014.07.003> Domain-Specific Languages and High-Level Frameworks for High-Performance Computing.
- [4] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE International Symposium on Workload Characterization (IISWC)*. 44–54. <https://doi.org/10.1109/IISWC.2009.5306797>
- [5] Valentin Churavy, Dilum Aluthge, Lucas C Wilcox, Simon Byrne, Maciej Waruszewski, Ali Ramadhan, Meredith, Simeon Schaub, James Schloss, Julian Samaroo, Jake Bolewski, Charles Kawczynski, Jeremy E Kozdon, Jinguo Liu, Oliver Schulz, Oscar, Pál Haraldsson, Takafumi Arakaki, and Tim Besard. 2022. JuliaGPU/KernelAbstractions.jl: v0.8.0. <https://doi.org/10.5281/zenodo.6324344>
- [6] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. 1989. An Efficient Method of Computing Static Single Assignment Form. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Austin, Texas, USA) (*POPL '89*). Association for Computing Machinery, New York, NY, USA, 25–35. <https://doi.org/10.1145/75277.75280>
- [7] Gregory Diamos, Andrew Kerr, Sudhakar Yalamanchili, and Nathan Clark. 2010. Ocelot: a dynamic optimization framework for bulk-synchronous applications in heterogeneous systems. In *2010 19th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 353–364.
- [8] Johannes Doerfert, Jose Manuel Monsalve Diaz, and Hal Finkel. 2019. The TRegion Interface and Compiler Optimizations for OpenMP Target Regions. In *OpenMP: Conquering the Full Hardware Spectrum - 15th International Workshop on OpenMP, IWOMP 2019, Auckland, New Zealand, September 11-13, 2019, Proceedings (Lecture Notes in Computer Science, Vol. 11718)*, Xing Fan, Bronis R. de Supinski, Oliver Sinnen, and Nasser Giacaman (Eds.). Springer, 153–167. https://doi.org/10.1007/978-3-030-28596-8_11
- [9] Johannes Doerfert and Hal Finkel. 2018. Compiler Optimizations for OpenMP. In *Evolving OpenMP for Evolving Architectures*, Bronis R. de Supinski, Pedro Valero-Lara, Xavier Martorell, Sergi Mateo Bellido, and Jesus Labarta (Eds.). Springer International Publishing, Cham, 113–127.
- [10] Johannes Doerfert and Hal Finkel. 2018. Compiler Optimizations for Parallel Programs. In *Languages and Compilers for Parallel Computing - 31st International Workshop, LCPC 2018, Salt Lake City, UT, USA, October 9-11, 2018, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 11882)*, Mary W. Hall and Hari Sundar (Eds.). Springer, 112–119. https://doi.org/10.1007/978-3-030-34627-0_9
- [11] Aleksandr Drozd. 2021. Benchmark. Online GitHub repository: <https://github.com/undertherain/benchmark/>, commit e1f22da320b0c7384cbd2f4df50255c7c2fa6b9d.
- [12] Peng Du, Rick Weber, Piotr Luszczek, Stanimire Tomov, Gregory Peterson, and Jack Dongarra. 2012. From CUDA to OpenCL: Towards a performance-portable solution for multi-platform GPU programming. *Parallel Comput.* 38, 8 (2012), 391–407. <https://doi.org/10.1016/j.parco.2011.10.002>
- [13] H Carter Edwards, Christian R Trott, and Daniel Sunderland. 2014. Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *Journal of parallel and distributed computing* 74, 12 (2014), 3202–3216.
- [14] Paul Feautrier and Christian Lengauer. 2011. Polyhedron Model. *Encyclopedia of parallel computing* (2011), 1581–1592.
- [15] Franz Franchetti, Tze Meng Low, Doru Thom Popovici, Richard M. Veras, Daniele G. Spampinato, Jeremy R. Johnson, Markus Püschel, James C. Hoe, and José M. F. Moura. 2018. SPIRAL: Extreme Performance Portability. *Proc. IEEE* 106, 11 (2018), 1935–1968. <https://doi.org/10.1109/JPROC.2018.2873289>
- [16] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. 1998. The Implementation of the Cilk-5 Multithreaded Language. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation* (Montreal, Quebec, Canada) (*PLDI '98*). Association for Computing Machinery, New York, NY, USA, 212–223. <https://doi.org/10.1145/277650.277725>
- [17] Fujitsu. 2022. https://github.com/fujitsu/dnnl_aarch64
- [18] Tobias Gysi, Christoph Müller, Oleksandr Zinenko, Stephan Herhut, Eddie Davis, Tobias Wicky, Oliver Fuhrer, Torsten Hoefer, and Tobias Grosser. 2021. Domain-Specific Multi-Level IR Rewriting for GPU: The OpenEarth Compiler for GPU-Accelerated Climate Simulation. *ACM Trans. Archit. Code Optim.* 18, 4, Article 51 (sep 2021), 23 pages. <https://doi.org/10.1145/3469030>
- [19] Ruobing Han, Jaewon Lee, Jaewoong Sim, and Hyesoon Kim. 2022. COX: CUDA on X86 by Exposing Warp-Level Functions to CPUs. *ACM Trans. Archit. Code Optim.* (jul 2022). <https://doi.org/10.1145/3554736>
- [20] Mark Harris et al. 2007. Optimizing parallel reduction in CUDA. *Nvidia developer technology* 2, 4 (2007), 70.
- [21] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 770–778.
- [22] J. A. Herdman, W. P. Gaudin, O. Perks, D. A. Beckingsale, A. C. Mallinson, and S. A. Jarvis. 2014. Achieving Portability and Performance through OpenACC. In *2014 First Workshop on Accelerator Programming using Directives*. 19–26. <https://doi.org/10.1109/WACCPD.2014.10>
- [23] Chuntao Hong, Dehao Chen, Wenguang Chen, Weimin Zheng, and Haibo Lin. 2010. MapCG: Writing Parallel Program Portable between CPU and GPU. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques* (Vienna, Austria) (*PACT '10*). Association for Computing Machinery, New York, NY, USA, 217–226. <https://doi.org/10.1145/1854273.1854303>
- [24] Intel. 2022. OneAPI Deep Neural Network Library (OneDNN). <https://github.com/oneapi-src/oneDNN>
- [25] Pekka Jääskeläinen, Carlos Sánchez de La Lama, Erik Schnetter, Kalle Raikila, Jarmo Takala, and Heikki Berg. 2015. pool: A performance-portable OpenCL implementation. *International Journal of Parallel Programming* 43, 5 (2015), 752–785.
- [26] Ralf Karrenberg and Sebastian Hack. 2012. Improving performance of OpenCL on CPUs. In *Compiler Construction*, Michael O'Boyle (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 1–20.
- [27] Andreas Klöckner. 2014. Loo.Py: Transformation-Based Code Generation for GPUs and CPUs. In *Proceedings of ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming (ARRAY'14)* (Edinburgh, United Kingdom). Association for Computing Machinery, New York, NY, USA, 82–87. <https://doi.org/10.1145/2627373.2627387>
- [28] Maria Kotsifakou, Prakalp Srivastava, Matthew D. Sinclair, Rakesh Kormarvelli, Vikram Adve, and Sarita Adve. 2018. HPVM: Heterogeneous

1156
1157
1158
1159
1160
1161
1162
1163
1164
1165
1166
1167
1168
1169
1170
1171
1172
1173
1174
1175
1176
1177
1178
1179
1180
1181
1182
1183
1184
1185
1186
1187
1188
1189
1190
1191
1192
1193
1194
1195
1196
1197
1198
1199
1200
1201
1202
1203
1204
1205
1206
1207
1208
1209
1210

- parallel virtual machine. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Vienna, Austria) (*PPoPP ’18*). Association for Computing Machinery, New York, NY, USA, 68–80. <https://doi.org/10.1145/3178487.3178493>
- [29] C. Lattner and V. Adve. 2004. LLVM: a compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004*. 75–86. <https://doi.org/10.1109/CGO.2004.1281665>
- [30] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2021. MLIR: Scaling Compiler Infrastructure for Domain Specific Computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2–14. <https://doi.org/10.1109/CGO51591.2021.9370308>
- [31] LLVM Contributors. 2021. OpenMP-aware optimizations. Online: <https://openmp.llvm.org/optimizations/OpenMPOpt.html>.
- [32] Simon Moll, Johannes Doerfert, and Sebastian Hack. 2016. Input Space Splitting for OpenCL. In *Proceedings of the 25th International Conference on Compiler Construction* (Barcelona, Spain) (*CC 2016*). Association for Computing Machinery, New York, NY, USA, 251–260. <https://doi.org/10.1145/2892208.2892217>
- [33] Sungdo Moon and Mary W Hall. 1999. Evaluation of predicated array data-flow analysis for automatic parallelization. *ACM SIGPLAN Notices* 34, 8 (1999), 84–95.
- [34] William Steven Moses. 2017. *How should compilers represent fork-join parallelism?* Master’s thesis. Massachusetts Institute of Technology.
- [35] William S. Moses, Lorenzo Chelini, Ruizhe Zhao, and Oleksandr Zinenko. 2021. Polygeist: Raising C to Polyhedral MLIR. In *2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 45–59. <https://doi.org/10.1109/PACT52795.2021.00011>
- [36] William S. Moses, Valentin Churavy, Ludger Paehler, Jan Hückelheim, Sri Hari Krishna Narayanan, Michel Schanen, and Johannes Doerfert. 2021. Reverse-Mode Automatic Differentiation and Optimization of GPU Kernels via Enzyme. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (St. Louis, Missouri) (*SC ’21*). Association for Computing Machinery, New York, NY, USA, Article 61, 16 pages. <https://doi.org/10.1145/3458817.3476165>
- [37] Cosmin E Oancea and Lawrence Rauchwerger. 2012. Logical inference techniques for loop parallelization. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*. 509–520.
- [38] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d’Alché-Buc, E. Fox, and R. Garnett (Eds.), Vol. 32. Curran Associates, Inc. <https://proceedings.neurips.cc/paper/2019/file/bdbca288fee7f92f2bfa9f7012727740-Paper.pdf>
- [39] Atmn Patel, Shilei Tian, Johannes Doerfert, and Barbara Chapman. 2021. A Virtual GPU as Developer-Friendly OpenMP Offload Target. In *50th International Conference on Parallel Processing Workshop* (Lemont, IL, USA) (*ICPP Workshops ’21*). Association for Computing Machinery, New York, NY, USA, Article 24, 7 pages. <https://doi.org/10.1145/3458744.3473356>
- [40] Matt Pharr and William R Mark. 2012. ispc: A SPMD compiler for high-performance CPU programming. In *2012 Innovative Parallel Computing (InPar)*. IEEE, 1–13.
- [41] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédéric Durand, and Saman Amarasinghe. 2013. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Seattle, Washington, USA) (*PLDI ’13*). Association for Computing Machinery, New York, NY, USA, 519–530. <https://doi.org/10.1145/2491956.2462176>
- [42] Mitsuhsisa Sato, Yutaka Ishikawa, Hirofumi Tomita, Yuetsu Kodama, Tetsuya Odajima, Miwako Tsuji, Hisashi Yashiro, Masaki Aoki, Naoyuki Shida, Ikuo Miyoshi, Kouichi Hirai, Atsushi Furuya, Akira Asato, Kuniki Morita, and Toshiyuki Shimizu. 2020. Co-Design for A64FX Manycore Processor and “Fugaku”. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–15. <https://doi.org/10.1109/SC41405.2020.000051>
- [43] Tao B Schardl, William S Moses, and Charles E Leiserson. 2019. Tapir: Embedding recursive fork-join parallelism into LLVM’s intermediate representation. *ACM Transactions on Parallel Computing (TOPC)* 6, 4 (2019), 1–33.
- [44] Adrian Schmitz, Julian Miller, Lukas Trümper, and Matthias S Müller. 2021. PPIR: Parallel Pattern Intermediate Representation. In *2021 IEEE/ACM International Workshop on Hierarchical Parallelism for Exascale Computing (HiPar)*. IEEE, 30–40.
- [45] Alexander Sergeev and Mike Del Balso. 2018. Horovod: fast and easy distributed deep learning in TensorFlow. <https://doi.org/10.48550/ARXIV.1802.05799>
- [46] Tyler Sorensen, Alastair F. Donaldson, Mark Batty, Ganesh Gopalakrishnan, and Zvonimir Rakamarić. 2016. Portable Inter-Workgroup Barrier Synchronisation for GPUs. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Amsterdam, Netherlands) (*OOPSLA 2016*). Association for Computing Machinery, New York, NY, USA, 39–58. <https://doi.org/10.1145/2983990.2984032>
- [47] Tyler Sorensen, Lucas F Salvador, Harmit Raval, Hugues Evrard, John Wickerson, Margaret Martonosi, and Alastair F Donaldson. 2021. Specifying and testing GPU workgroup progress models. *Proceedings of the ACM on Programming Languages* 5, OOPSLA (2021), 1–30.
- [48] George Stelle, William S. Moses, Stephen L. Olivier, and Patrick McCormick. 2017. OpenMPIR: Implementing OpenMP Tasks with Tapir. In *Proceedings of the Fourth Workshop on the LLVM Compiler Infrastructure in HPC* (Denver, CO, USA) (*LLVM-HPC’17*). Association for Computing Machinery, New York, NY, USA, Article 3, 12 pages. <https://doi.org/10.1145/3148173.3148186>
- [49] George Stelle, William S. Moses, Stephen L. Olivier, and Patrick McCormick. 2017. OpenMPIR: Implementing OpenMP Tasks with Tapir. In *Proceedings of the Fourth Workshop on the LLVM Compiler Infrastructure in HPC* (Denver, CO, USA). ACM, New York, NY, USA, Article 3, 12 pages. <https://doi.org/10.1145/3148173.3148186>
- [50] John A. Stratton, Vinod Grover, Jaydeep Marathe, Bastiaan Aarts, Mike Murphy, Ziang Hu, and Wen-mei W. Hwu. 2010. Efficient Compilation of Fine-Grained SPMD-Threaded Programs for Multicore CPUs. In *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization* (Toronto, Ontario, Canada) (*CGO ’10*). Association for Computing Machinery, New York, NY, USA, 111–119. <https://doi.org/10.1145/1772954.1772971>
- [51] John A. Stratton, Sam S. Stone, and Wen-mei W. Hwu. 2008. MCUDA: An Efficient Implementation of CUDA Kernels for Multi-core CPUs. In *Languages and Compilers for Parallel Computing*, José Nelson Amaral (Ed.). Vol. 5335. Springer, Berlin, Heidelberg, 16–30. https://doi.org/10.1007/978-3-540-89740-8_2 Series Title: Lecture Notes in Computer Science.
- [52] Sander Stuijk, Marc Geilen, and Twan Basten. 2006. Sdf³: Sdf for free. In *Sixth International Conference on Application of Concurrency to System Design (ACSD’06)*. IEEE, 276–278.

- 1321 [53] Zehra Sura, Xing Fang, Chi-Leung Wong, Samuel P. Midkiff, Jaejin Lee, 1376
1322 and David Padua. 2005. Compiler Techniques for High Performance 1377
1323 Sequentially Consistent Java Programs. In *Proceedings of the Tenth ACM 1378*
1324 *SIGPLAN Symposium on Principles and Practice of Parallel Programming* 1379
1325 (Chicago, IL, USA) (*PPoPP '05*). Association for Computing Machinery, 1380
1326 New York, NY, USA, 2–13. <https://doi.org/10.1145/1065944.1065947> 1381
1327 [54] Ximin Tian, Hideki Saito, Ernesto Su, Jin Lin, Satish Guggilla, Diego 1382
1328 Caballero, Matt Masten, Andrew Savonichev, Michael Rice, Elena 1383
1329 Demikhovsky, Ayal Zaks, Gil Rapaport, Abhinav Gaba, Vasileios Por- 1384
1330 podas, and Eric N. Garcia. 2017. LLVM Compiler Implementation for 1385
1331 Explicit Parallelization and SIMD Vectorization. In *Proceedings of the* 1386
1332 *Fourth Workshop on the LLVM Compiler Infrastructure in HPC*. ACM, 1387
1333 Denver, CO, USA, 4:1–4:11. <https://doi.org/10.1145/3148173.3148191> 1388
1334 [55] Nicolas Vasilache, Benoit Meister, Muthu Baskaran, and Richard Lethin. 1389
1335 2012. Joint scheduling and layout optimization to enable multi-level 1390
1336 vectorization. *IMPACT* 12 (2012). 1391
1337 [56] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya 1392
1338 Goyal, Zachary Devito, William S. Moses, Sven Verdoolaege, Andrew 1393
1339 Adams, and Albert Cohen. 2019. The Next 700 Accelerated Layers: 1394
1340 From Mathematical Expressions of Network Computation Graphs to 1395
1341 Accelerated GPU Kernels, Automatically. *ACM Trans. Archit. Code 1396*
1342 Optim.
- 1343 16, 4, Article 38 (oct 2019), 26 pages. <https://doi.org/10.1145/3355606> 1397
1344 [57] Oleksandr Zinenko, Sven Verdoolaege, Chandan Reddy, Jun Shirako, 1398
1345 Tobias Grosser, Vivek Sarkar, and Albert Cohen. 2018. Modeling the 1399
1346 Conflicting Demands of Parallelism and Temporal/Spatial Locality in 1400
1347 Affine Scheduling. In *Proceedings of the 27th International Conference 1401*
1348 on Compiler Construction (Vienna, Austria) (*CC 2018*). Association for 1402
1349 Computing Machinery, New York, NY, USA, 3–13. <https://doi.org/10.1145/3178372.3179507> 1403
1350
1351
1352
1353
1354
1355
1356
1357
1358
1359
1360
1361
1362
1363
1364
1365
1366
1367
1368
1369
1370
1371
1372
1373
1374
1375